NASA Contractor Report 181938

Investigation of the Applicability of a Functional Programming Model to Fault Tolerant Parallel Processing for Knowledge-Based Systems

Richard Harper

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

Contract-NAS1-18565
December 1989

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

NASA Contractor Report 181938

Investigation of the Applicability of a Functional Programming Model to Fault Tolerant Parallel Processing for Knowledge-Based Systems

Richard Harper

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

Contract-NAS1-18565
December 1989

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Table of Contents

ABSTRACT

In a fault-tolerant parallel computer, a functional programming model can facilitate distributed checkpointing, error recovery, load balancing, and graceful degradation. Such a model has been implemented on the Draper Fault Tolerant Parallel Processor (FTPP). When used in conjunction with the FTPP's fault detection and masking capabilities, this implementation results in a graceful degradation of system performance after faults. Three graceful degradation algorithms have been implemented and are presented. A user interface has been implemented which requires minimal cognitive overhead by the application programmer, masking such complexities as the system's redundancy, distributed nature, variable complement of processing resources, load balancing, fault occurrence and recovery. This user interface is described and its use demonstrated. The applicability of the functional programming style to the Activation Framework, a paradigm for intelligent systems, is then briefly described.

0. INTRODUCTION

Future autonomous and semi-autonomous applications such as the Space Station Thermal Management System [NA86], the Adaptive Tactical Navigator [Jo85], and others will require the use of "intelligent" or "knowledge-based" systems to execute real-time, mission- or life-critical functions. In addition to high reliability requirements, it is predicted that these functions will require computational throughput in excess of that achievable by advanced uniprocessors, thus mandating the use of a parallel processing system.

In parallel computer architectures, there is a high likelihood that at any given time a part of the system will exhibit faulty behavior. The ability to tolerate this behavior must be an integral feature of such architectures and their programming models[1]. For example, the number of processors available in a parallel system at a given time varies as failures occur. Programming models which cannot accommodate a variable number of processors require a spare processor for each failure to be tolerated. After the spares are exhausted, additional failures render the system unusable. A programming model which can accommodate a variable number of processors during the execution of a computation allows graceful degradation of system performance as failures occur, while allowing use of the spares to increase

[1] A programming model is a paradigm which represents to a programmer the way in which a computer will execute a program. Traditional uniprocessors use the Von Neumann model of sequential program execution.

system performance prior to failures. A system which degrades gracefully has an automated, orderly process for dealing with hardware failures as they occur. Such a process results in minimal disruption of an application program executing on the system while the system responds to the failure.

To facilitate the discussion of graceful degradation, some nomenclature pertinent to the topic needs to be introduced. The computational state of a program is a body of data which must persist over an extended period of time during program execution. It performs the same function that memory does for a human being, reminding the program of the conditions under which it is operating. One way humans have of dealing with imperfect memories is to write down a copy of some information for future reference, to be used in case of an incident of absent-mindedness. In a computer, the process of making a backup copy of state information is called checkpointing. When something goes wrong with a program during its execution, it may be possible to retry a computation, perhaps in another processor, by using the checkpointed state of the program, saved before the failure was detected. The program is in effect "rolled back" to an earlier state. When programs are parallelized and distributed over several computers, a rollback in one machine may trigger a rollback in another, resulting in an undesirable domino rollback effect.

To degrade gracefully, a parallel system requires a means of checkpointing and backward recovery which does not depend upon global coordination of checkpoint placement [Ho83] and which will not result in domino rollback [Ku86]. Furthermore, optimal performance requires maximum utilization of system resources, requiring in turn a means of balancing the load across all the processors comprising the system. A primary problem in implementing load balancing is the transport of large amounts of computational state. A programming model which allows the concise representation of large amounts of computational work would facilitate the implementation of an efficient load balancing scheme. Load balancing can also be used to facilitate failure recovery algorithms by evacuating work from degraded processing sites, that is, redundant processing sites which because of failures possess insufficient redundancy to support the application's required reliability.

Parallel algorithms can cause saturation of system resources because of excessive runtime generation of parallelism [Tr87]. A solution requires a portion of the extant modules to be aborted to free resources and allow the remaining extant modules to spawn the children required for their completion. Subsequently the aborted modules can themselves be restarted and can execute to completion. This is a form of rollback and also requires a form of distributed checkpointing and recovery.

A programming model and the operating system which supports it must facilitate development of parallel algorithms, while masking the distributed and fault-tolerant nature of the underlying system from the application programmer. Minimization of irrelevant cognitive noise eases coding, testing, and validation.

The referential transparency inherent in a *functional programming model* holds promise as a partial solution to these problems. Such a model has been implemented on the Charles Stark Draper Laboratory (CSDL) Fault Tolerant Parallel Processor (FTPP). The FTPP supports Byzantine Resilient processing sites each of which is capable of detecting and masking faults with near-unity probability. When used in conjunction with the FTPP's fault detection and masking capabilities, the functional programming style can facilitate distributed checkpointing, error recovery, load balancing, and graceful degradation.

Another programming paradigm which has been proposed for real-time, intelligent, i.e., knowledge-based systems, is the Activation Framework [Gr85]. The activation framework parallelizes an application as a set of communicating experts. Since each "expert" possesses persistent state, this paradigm is not functional. Nevertheless, it is a model intrinsically well-suited to the architecture of the FTPP, especially for knowledge-based applications requiring high levels of reliability. Implementing a system to support AF constructs on the FTPP would provide an informative non-functional counterpart to the functional programming model examined in this study.

This report begins with an overview of the FTPP architecture, followed by a description of the implementation of a Remote Procedure-based functional programming model on the FTPP. A user interface has been implemented which requires minimal cognitive overhead by the application programmer, masking such complexities as the system's redundancy, distributed nature, variable complement of processing resources, load balancing, fault occurrence, and recovery. This user interface is described and its use demonstrated. Three graceful degradation algorithms are described and a preliminary evaluation is provided. The applicability of the functional programming style to the Activation Framework intelligent system paradigm is then briefly described.

1. FAULT TOLERANT PARALLEL PROCESSOR OVERVIEW

The testbed for the implementation and evaluation of the functional model is the prototype Fault Tolerant Parallel Processor (FTPP), a high-reliability, high-throughput parallel processor under development at CSDL. The FTPP achieves high throughput by using a

multiplicity of loosely-coupled Processing Elements (PEs) which communicate via message-passing over a shared communication medium. The FTPP achieves high reliability by being capable of surviving a specified number of component failures with a probability approaching unity.

A conservative failure model is to consider failures as consisting of arbitrary or even malicious behavior on the part of failed components. This type of fault, known as a *Byzantine fault*, may include stopping and then restarting execution at a future time, sending conflicting information to different destinations, and other types of malicious behavior. While certainly not common, Byzantine failures cannot be ignored in the design of fault-tolerant computers for critical applications. For example, at least one inflight failure of a triplex digital computer system was traced to a Byzantine fault and the lack of appropriate architectural safeguards against such faults [La86]. In the Fault Tolerant MultiProcessor (FTMP), a failure caused one channel to send conflicting interpretations of faulty behavior to other channels. The list goes on, and would be longer if other architectures were capable of tolerating and logging Byzantine behavior. Because such failure modes clearly exist in practice, an ultra-high reliability system must be able to tolerate them.

Among other requirements [Ha87], Byzantine Resilience requires that PEs be replicated and synchronously execute functionally identical code on bitwise-identical inputs. Fault masking and detection must be obtained via bitwise comparison of outputs from the redundant processing group. In the FTPP, PEs are connected to special-purpose Network Elements (NEs) which permit inter-PE communication both for fault tolerance-related purposes (i.e., distribution of input data, voting of output data, and synchronization of redundant groups) and inter-redundant group purposes (i.e., message passing in a parallelized application). Figure 1 shows one possible arrangement of NEs and PEs into a 16-PE, 4-NE "cluster". The NEs in the cluster are fully connected to each other via point-to-point communication links, which also serve as physical fault isolation barriers. Inter-NE links are used for interprocessor communication and synchronization, and are the only physical connections between primary fault containment regions. Each NE also possesses a port to each of its subscriber PEs. An NE and its associated PEs comprise a fault containment region. Consequently, a Byzantine Resilient Virtual Group (VG) must comprise at least three PEs each subscribing to a unique NE. Figure 1 shows a mixed redundancy configuration of the cluster. In this example, the PEs of the cluster are arranged into one quadruply redundant virtual group (VG) Q1, one triply redundant VG T1, and nine simplex VGs S1 through S9. As an example of redundancy management policies possible

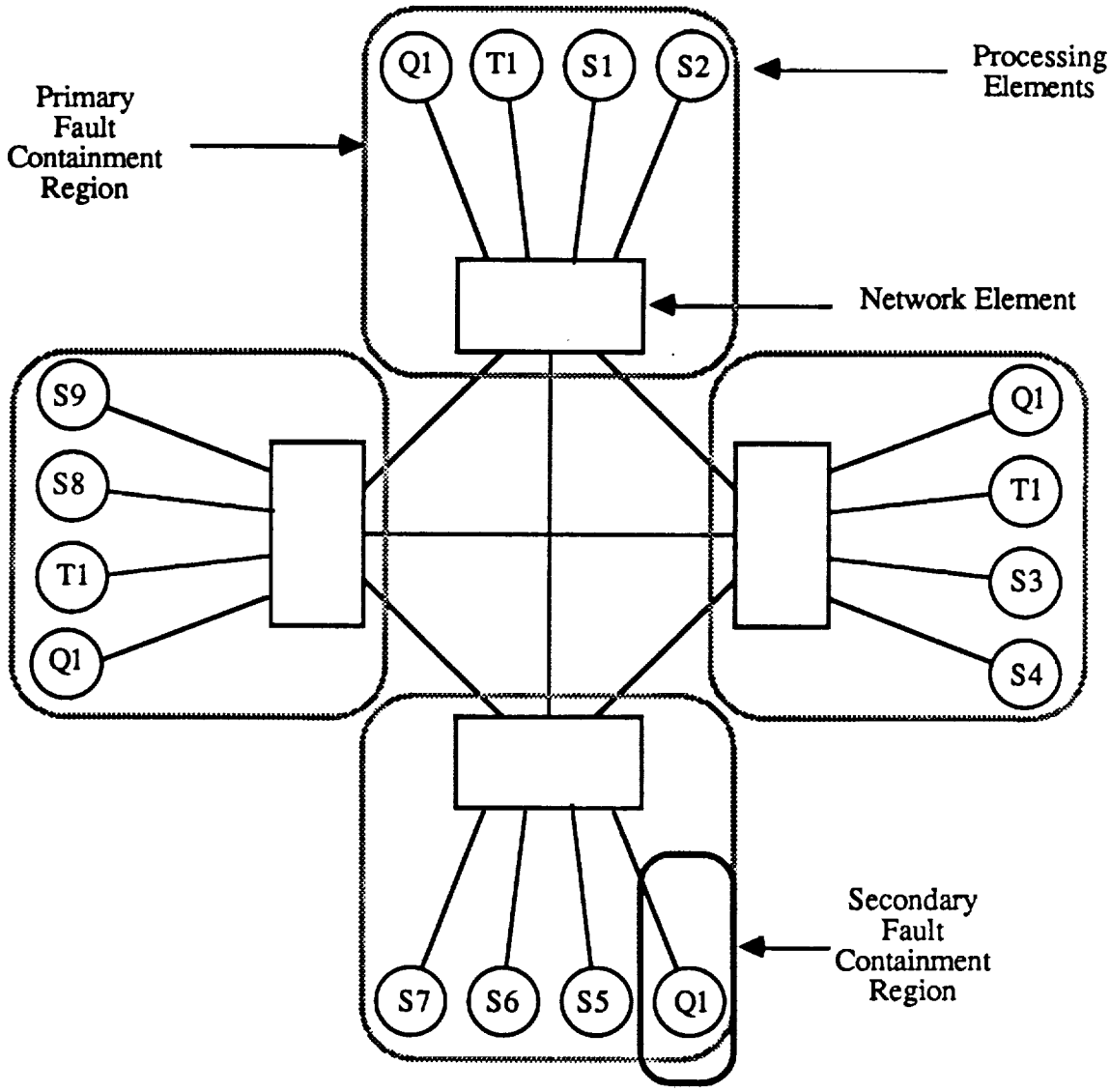

Figure 1. Fault Tolerant Parallel Processor Cluster

within this cluster, if a member of quad Q1 fails, then simplex S5 may be assigned to Q1 to restore its redundancy, as shown in Figure 2. Alternatively, Q1 may be disbanded and its former members used as spares in a graceful degradation scheme, or it may be designated to be a triplex VG. Numerous other redundancy configurations and redundancy man-

agement strategies are possible with this cluster. Algorithms required for reconfiguration have been developed and demonstrated on the prototype cluster.

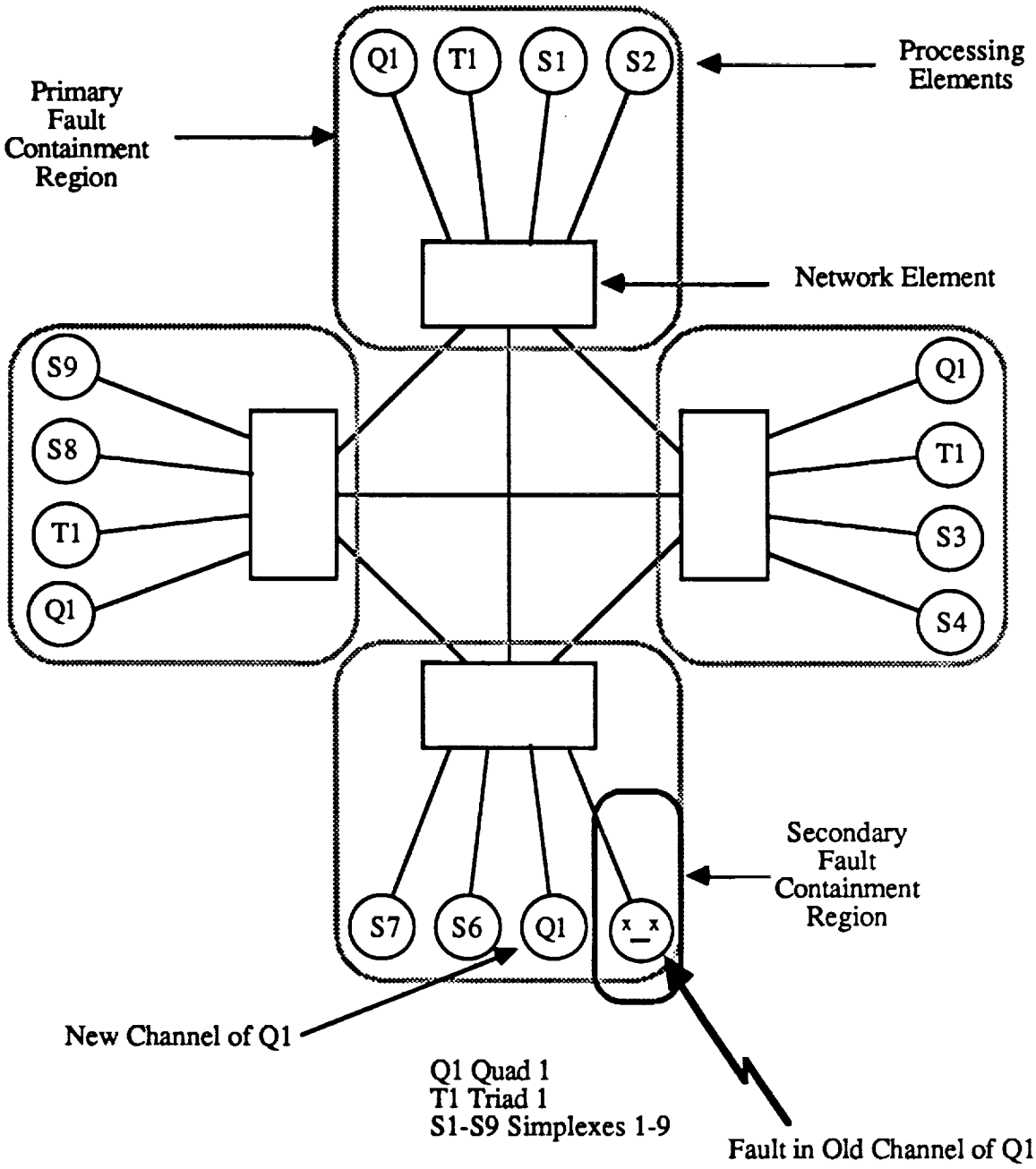

Figure 2. Repair of Quadruplex Virtual Group Q1

Because of limited NE execution speed and communication link bandwidth, all of the PEs of a large ensemble cannot be efficiently supported by a single cluster. An FTPP ensemble is therefore assembled from several clusters, as shown in Figure 3. The size of each

cluster is optimized for the reliability and performance parameters of importance in a given application. Reliability and performance formulations have been developed which help determine this optimum [Ha87]. Given clusters of such an optimal size, specialized *Input/Output Elements* (IOEs) are used for their interconnection. One IOE subscribes to each NE and its sole function is inter-cluster communication. Fault-masking intercluster communication is achieved by the use of redundant intercluster links connected between the IOEs of different clusters.

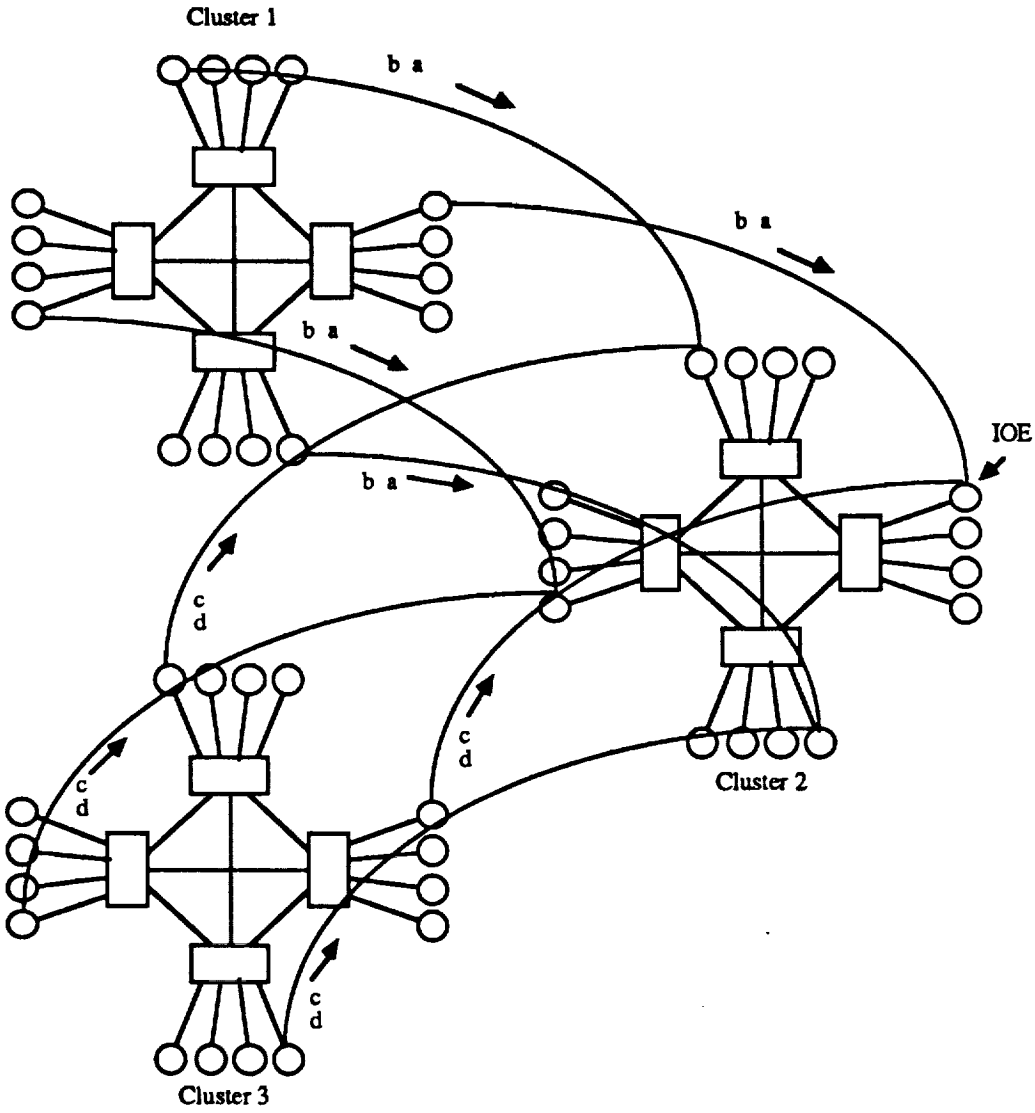

Figure 3. FTPP Composed of Three Clusters

Regardless of the redundancy level of the VG complement, the FTPP's fault tolerance is transparent to the applications programmer. The FTPP is programmed as a non-redun-

dant parallel processor which is extremely reliable. To assist in this, Byzantine Resilient inter-VG communication protocols have been developed which guarantee that messages are delivered in the order sent, and that messages sent to one or more VGs are received in the same order everywhere. When the number of simultaneously active failures does not exceed the design limit, such guarantees obviate consideration of uncontrollably faulty component behavior by the designers of distributed applications.

A 16-PE prototype FTPP cluster is in an advanced stage of integration at the Draper Laboratory. The cluster includes the PEs, the NEs, the operating system, reconfiguration functions, the functional programming model developed under this Task, and several demonstration applications. Currently, evaluation and optimization of the performance of this prototype is underway. A second 32-PE cluster is in the planning stages. Further information on the FTPP is available in [Ha87], [Ha88] and [Ab88].

2. FUNCTIONAL PROGRAMMING

2.1 OVERVIEW

A functional programming model has been shown to provide benefits for both system reliability [Li86] and for system performance in parallel processing architectures [Fa85], [Ve84]. A functional programming model structures a computation into the evaluation of its constituent functions. The result of a given evaluation is uniquely determined by the arguments given the function. The only information exchange between the function and its caller are the initial passing of parameters and the return of the result. This property of "determinism" or "referential transparency" has been proposed to simplify error recovery in fault-tolerant systems [Li86], [Ja86].

Because of referential transparency the result produced by any function is insensitive to the processor on which it is executed in the absence of failures. As the system degrades due to failures, function instantiations can be redirected to nonfaulty processors with a commensurate reduction in ensemble throughput. The functional programming model thereby allows graceful degradation by being relatively insensitive to the number of available processors in the ensemble, assuming faults can be detected and survived.

The concise representation of a function by its name, input arguments, and caller's name allows migration of functions to other processors without the need to transfer massive amounts of state information. Load balancing to maximize utilization of system resources can thus be provided without substantial penalty.

Function invocation constitutes a natural rollback point. Rollback can be the result of the failure of a processor, or the unwinding of a calling tree in a deadlock-recovery algorithm. The function to be computed, its arguments, and the destination of the function's result are all the information necessary to restart the function, and they can be saved by the function's caller at low cost. If rollback is necessary, the function can be restarted on any processor by transferring the appropriate information to that processor. Because of referential transparency, the result of the restarted computations is identical to that of the original computation, merely delayed in time.

Functional programs are typically computationally intensive and written at a high level of abstraction [Ve84]. It is undesirable to complicate the programming problem with details of distributed system operation such as the mapping of functions onto physical processing sites or arranging for the transfer of arguments from parent to child. Furthermore, the dynamic nature of a fault-tolerant system as it responds to failures should be transparent to applications. An appropriately low level of cognitive overhead can be provided which allows a programmer to work with functional abstractions and which hides the details of the underlying hardware and system operation.

2.2 THE REMOTE PROCEDURE MODEL

An operating system which supports parallel execution of applicative functional programs and provides transparent fault detection, fault masking and error recovery with near unity probability has been implemented on the FTPP. The functional abstraction used by the application programmer is that of a functional Remote Procedure (RP). Unlike functional programming languages which force the programmer to adhere to the requirements of referential transparency, the RP is an atomic unit of computation which can be coded in any standard programming language, assuming the RP is free of side effects outside its own variable scoping. C and Ada have been used in demonstration programs. Although the results of a given RP depend only on its arguments, it may have internal state and may access a shared static environment. RPs call other RPs in much the same way that a function calls another function. However, the caller is not suspended or blocked while waiting for the called function to complete. The called function may proceed in parallel with the caller as well as any other function in the system. A given RP may invoke a number of children and then await the completion of none, one, some, or all of its children before returning to its parent. As the program executes, the computation tree expands and contracts as necessary as functions recursively create children and consume their computational results.

Within the FTPP, RPs are balanced evenly among the non-faulty VGs for maximal resource utilization by a distributed system process called the RP Manager, whose operation is described below. The RP Manager transfers the argument and response between the VGs on which the parent and child are instantiated. However, the programmer is not concerned with which VG executes an RP. His conceptualization of his program is that of a directed computation tree, beginning with a main RP, called the "root RP", and expanding and contracting as necessary to execute his parallelized algorithm, as if each RP will execute on a different processor.

At the lowest level, an RP executes as an imperative task on a VG within the FTPP. A parallel program may consist of one or many different RPs of which many instantiations may exist at a given time. Because of the large memory size of the FTPP's processing elements, the code for all RPs can be resident on all VGs in the system. Load balancing therefore involves only the transfer of data. The code for the various RPs is installed on the system by normal compiling, linking, and downloading operations. While members of a given VG must execute identical load modules for fault tolerance reasons, it is possible to install different versions of code on different VGs to minimize memory utilization. This has not yet been necessary. If heterogeneous load modules become necessary, our current scheme allows load balancing only to a VG which possesses the appropriate code for the RP being migrated. Another possibility is to develop a scheme to move code from one VG to another.

2.3 THE USER INTERFACE TO THE RP MODEL

Remote Procedures are created and controlled by the programmer using several calls (Figure 4). RPs have dual personalities; they operate both as parents and as children. The system calls therefore form two distinct groups. Five calls are provided for the use of children. When an RP begins to execute, it calls `rp_myargs()` to obtain a pointer to its arguments. `Rp_myrsp()` returns a pointer to memory in which the RP is to write its return value upon completion. If the RP has no further use for its arguments and wishes to return the memory allocated for their use to the system, it can do so by calling `rp_relargs()`. When the RP is completed and the return value has been written to the designated response area, the RP calls `rp_cmpltd()`. The response is then returned to the parent and the child is terminated. Some of the calls explicitly deschedule the calling RP. One such call is `rp_susp()`. It causes the calling RP to be suspended, explicitly returning control to the

system. The RP will be not be resumed until the next scheduled iteration of the RP Manager.

The remaining calls are provided for the use of RPs operating as parents. To invoke a child, the parent calls `rp_create()` with the identifier of the RP it wishes to start. When resources do not permit the creation of this child, an error is returned. Otherwise, the call returns a receipt to the parent containing a system-wide unique identifier for this child. The parent uses this receipt to query the system for information about this child or to otherwise control the activities of the child. The parent informs the RP Manager that the arguments for a given child have been initialized by calling `rp_start()`. If the RP Manager returns an error condition from the child's execution, the parent may restart the child by calling `rp_start()` again. If the parent no longer has a need for the result to be returned by a given child or if it wishes to free up resources to allow the creation of other children, it calls `rp_release()` with the receipt of the child it intends to terminate. If it wants to terminate all of its children, it calls `rp_flush()`. `Rp_next()` and `rp_nextb()` return information to a parent about the status of its children. `Rp_next()` returns the receipt of the oldest child which has completed but whose result has not been read. If no child is completed, it returns an indication that no child has returned its result yet. `Rp_nextb()` suspends the parent until the next child returns. When called with the receipt of a child, `rp_read()` returns a pointer to a result if the child has completed; otherwise it returns a null pointer. `Rp_readb()` blocks the parent until the specified child has returned its result.

System Calls Used by RPs when Operating as a Parent
`rp_create()`
`rp_start()`
`rp_release()`
`rp_flush()`
`rp_next()`
`rp_nextb()`
`rp_read()`
`rp_readb()`

System Calls Used by RPs when Operating as a Child
`rp_myargs()`
`rp_myrsp()`
`rp_relargs()`
`rp_compltd()`
`rp_susp()`

Figure 4. Remote Procedure System Calls

The use of these system calls is illustrated by the following example. The most expensive portion of AI programs is usually some type of search [Ki85]. While many searches are conducted over an inherently parallel domain, A* searches, which attempt to find the

least costly route through some problem space, have been most successfully parallelized on shared-memory architectures. A* search algorithms which are designed for architectures similar to that of the FTPP, i.e. which have distributed memory and processors which communicate via message passing, have not in general produced significant performance gains [Ku88]. Frequently, however, applications can tolerate a less-than-optimal solution. In fact, some time-sensitive applications require only the best solution which can be arrived at in a given amount of time. In these cases, it is often possible to trade quality of solution for speed of computation. Speed is increased in A* searches by limiting the number of nodes which are expanded as the search progresses. Decisions about node expansion are based on a cost function, which includes a heuristic component which can be made artificially high by imposing some external constraint [Pe84].

The RP model can be used to advantage in these situations by allowing many searches with different heuristics to proceed until the time constraint is reached, choosing the best solution from those which have completed, and killing any uncompleted searches. Figure 5a shows the root RP of this computation. It spawns a set of children, each of which conducts an A* search using a different constraint to generate the value of the heuristic evaluation function. As children complete, the solutions they return are added to a list which is sorted by the cost of the solution. Children are created with calls to `rp_create()` and `rp_start()`. By calling `rp_next()`, the root RP obtains the RP identifier of the oldest child to complete. If none have completed, it suspends itself, awaiting resumption by the RP Manager. At that point it reads the result of the completed child, inserts it at the correct place in the sorted list of completed children, and decides whether it can wait for any more outstanding children to complete. If not, it uses the best result so far and kills all remaining children by calling `rp_kill()`.

Figure 5b shows the child RP as it obtains its arguments by calling `rp_myargs()`, conducts its search, gets memory space in which to write its results by calling `rp_myrsp()`, and returns its result to the parent by calling `rp_cmpltd()`.

While these calls require the application programmer's awareness of the parallel nature of the program, they hide the system's hardware redundancy, failure behavior and recovery, mapping of RP executions to processors, and load balancing.

```
root_rp()
{
 /*  Variable Declarations */
 extern func_id_struct *astar_search;
 struct solution_struct solutions[num_kids], *solution,
 best_solution;
 struct arg_struct *arg;
 int start_time, rp_id, arg_addr;
 ...

 start_time = sys_time();

 /*  Create and start children */
 for(num_kids = 0; num_kids < max_num_kids; num_kids++)
 {
  /* Obtain RP ID and space for arguments. */
  if(! (rp_create(astar_search, &arg_addr, &rp_id) )
     break;
  else
  {
   /* Initialize the arguments. */
   arg = (struct arg_struct *) arg_addr;
   init_A*_nodes(arg);
   arg->constraint = max_constraint - (num_kids * delta);

   /* Start child to conduct search. */
   rp_start(rp_id);
  }
 }
 /*
     Read solutions as they arrive until real-time constraint
     is reached. Suspend if no children have completed.
 */
 while( (sys_time() - start_time < timeout) && num_kids >= 0)
 {
  if(rp_next(&rp_id) )
  {
   solution = (struct solution_struct *) rp_read(rp_id);
   sort_solutions(solution, solutions);
   num_kids--;
  }
  else rp_susp();
 }
 /*
     When real-time constraint arrives, flush all extant
     children and select best solution.
 */
 if (num_kids ≥ 0)
  rp_flush();
 best_solution = least_cost_solution(solutions);
}
```

Figure 5a. Root RP used to obtain the best solution to an A* search given a real-time constraint.

```
astar_search()
{
 /*  Variable Declarations */
 struct arg_struct *arg;
 struct solution_struct *solution;

 /*  Obtain Arguments */
 arg = (struct arg_struct *) rp_myargs();

 /*  Obtain memory to hold solution */
 solution = (struct solution_struct *) rp_myrsp();

 /*  Conduct A* search */
 astar(&arg->start_node, &arg->end_node, arg->constraint,
 solution);

 /*  Return result to parent and terminate. */
 rp_cmpltd();
}
```

Figure 5b. RP used to conduct an A* search given a constraint which produces a less than optimal solution based on a heuristic evaluation function.

2.4 FTPP OPERATING SYSTEM

The FTPP's Operating System (FTPPOS) comprises a non-preemptive multitasking scheduler and a set of system services which include Fault Detection, Identification, and Reconfiguration (FDIR) and RP management. FDIR is a process which detects an error in the system, identifies the faulty component and reconfigures the system to expunge the faulty component. FDIR is also responsible for notifying the RP management software that a VG is degraded and must be evacuated of computational load. The RP Manager is composed of four major functions: scheduling of RPs, supporting the user interface, deadlock detection and recovery, and supporting system reconfiguration to effect graceful degradation.

2.5 IMPLEMENTATION DETAILS OF THE RP MANAGER OPERATION

2.5.1 RP MANAGER AS A DISTRIBUTED PROCESS

The RP Manager is a distributed process which controls the scheduling and execution of RPs generated by the user's computation. An instance of the RP Manager is resident on every VG in the system. Each instance communicates with its peers by means of message passing. They cooperate to handle the calls described in §2.3, schedule RP execution, distribute the RP load evenly among the VGs, detect and resolve deadlock, and respond to FDIR reconfiguration directives by transferring RPs from degraded VGs to healthy ones.

2.5.2 DATA STRUCTURES USED BY THE RP MANAGER

The RP Manager uses several data structures to support its operation. One structure, an instance of which is resident on each VG, contains a system-wide unique set of reusable RP identifiers. The structures which support scheduling and load balancing are the pending list and the execution list. The reciprocity between entries in the pending and execution lists is shown in Figure 6. The latter is further partitioned into a ready queue, a balance queue, and a suspend queue.

2.5.2.1 RP IDENTIFIERS

To ensure that time delays associated with RP responses do not cause messages latent in the system to arrive unexpectedly and be confused with expected messages from other RPs, each RP is assigned a system-wide unique identifier at the time it is invoked. The *RP identifier* is considered a finite resource and presently each VG can generate 128 unique identifiers[2]. At system startup, the RP identifiers on a VG are linked into a list of free identifiers. As RP invocations are accepted, identifiers are removed from the list. When RPs complete, their identifiers are returned to the free list. When the free list is empty, `rp_create()` calls are denied. The RP identifier is also used as an index into the VG's pending list, pointing to the entry for that RP. When it is instantiated, i.e. scheduled for execution, each RP is given a second system-wide unique identifier, called the *instant identifier*. This identifier is similar to that of the RP identifier, except that it functions as an index of its entry in the execution list of the host VG.

2.5.2.2 THE PENDING LIST

Each VG hosts a *pending list* of all the children whose parents are instantiated on that VG. Each entry contains sufficient information to terminate or restart the child, and is one side of the link between parent and child in the system. As an RP fulfills its destiny, various fields in the pending list entry are updated.

2.5.2.3 THE EXECUTION LIST

The other half of the parent-child link is the *execution list* entry resident on the VG which actually instantiates the child RP. The execution list is a record of the RP requests received by the RP Manager on a given VG. The first such request is made with a parent's

---

[2]This is an arbitrary constraint made for system programming convenience and can be easily altered.

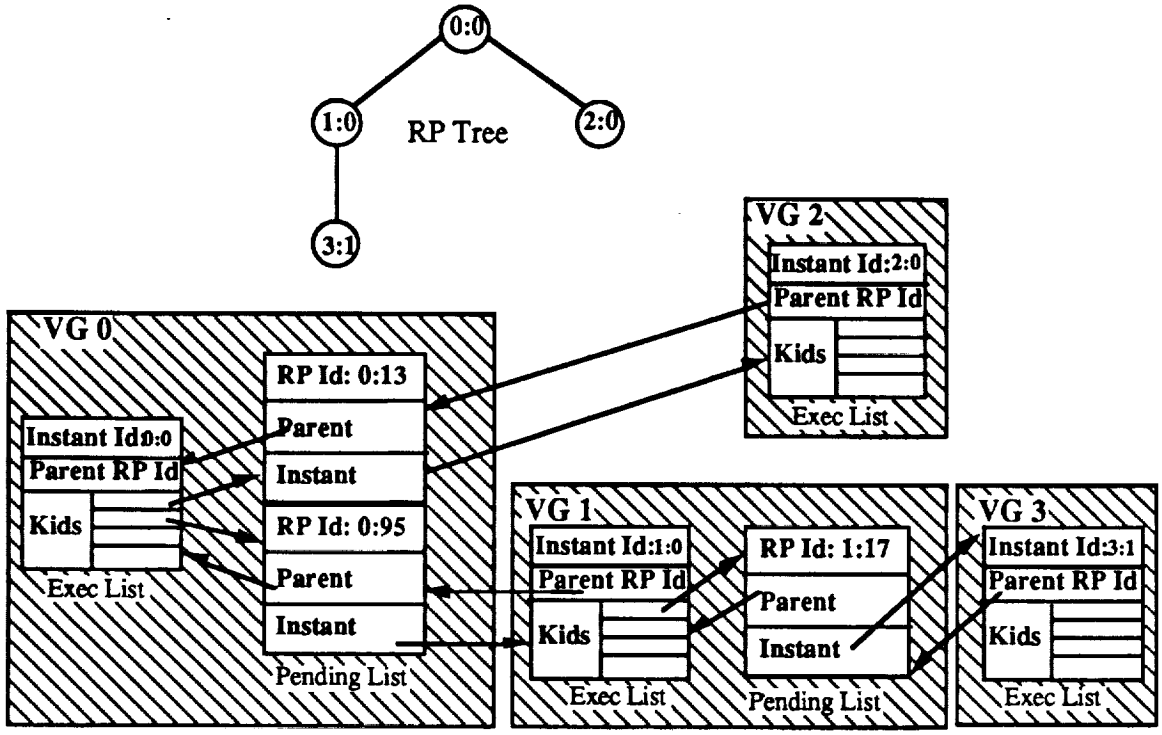

Figure 6. Pending and Execution Lists

call to `rp_start()`. The entries in the execution list are divided into three prioritized queues: the *balance queue*, the *ready queue*, and the *suspend queue*.

The balance queue contains RPs which have not yet been instantiated on a VG. They have not been allocated memory for arguments and responses, and they have not been started. These RPs can be transferred to another VG with very little communication overhead, and their presence on the balance queue implies that they are candidates for load balancing. Once an RP is instantiated, it acquires state and the transfer of this state information is much more costly. In the present system a "pseudo-transfer" of instantiated RPs has been implemented to support one the the graceful degradation algorithms (see Section 3). However, RPs transferred to balance load are uninstantiated.

When the RP Manager processes a call to `rp_start()`, it allocates an execution list entry for the designated RP and places the RP on the balance queue. The balance queue is prioritized by generation. If the load balancing logic within the RP Manager decides to transfer an RP in its balance queue to another VG, it removes the RP from its execution list. The RP Manager on the receiving end of the transaction places the transferred RP on the balance queue of its own execution list.

The *suspend queue* contains RPs which have been instantiated but have suspended themselves or have been suspended by the RP Manager pending the occurrence of some event such as child completion. Once the awaited event has occurred, the RP is transferred to the ready queue.

The *ready queue* contains RPs which can be run but have been implicitly suspended by virtue of making a system call. In this way, all RPs are given fair access to processing time on their VGs. The *ready queue* is sorted by the generation of the RPs in the queue. RPs with a higher generation number, i.e. younger RPs, are at the head of the queue and therefore will execute before older RPs. This prioritization is based on the assumption that in most cases children must complete before their parents can complete.

2.5.2.4 MEMORY ALLOCATION STRUCTURES AND ALGORITHMS

Memory for arguments and responses is allocated in fixed size blocks. More sophisticated schemes exist which use memory more efficiently while paying a performance penalty for this conservation, and this could form a topic for future development. All of the free memory in the system is used for RP arguments and responses. A linear equation is solved which determines the maximum number of RPs which can be instantiated on each VG based on the known argument/response size and the memory requirements of other FTPPOS processes.

2.5.3 LOAD BALANCING

The work load of a VG is defined as the set of RPs in its execution list. Parallel computation is achieved by distributing these RPs evenly among the non-failed VGs in the FTPP. In our implementation, load balancing also supports fault tolerance strategies for graceful degradation and for deadlock detection and recovery.

Load balancing requires a certain amount of overhead. Since a primary objective of inducing parallelism is to improve performance, a load balancing algorithm has been devised which minimizes this overhead while remaining responsive to changing system state. Migratable RPs have not yet begun to execute and reside on a balance queue. Thus, only arguments and responses need to be transferred to run a child RP on a VG remote from its parent. When an RP is balanced away, it is chosen from the head of the balance queue, i.e. an "older" RP is chosen. This results in rapid distribution of a significant amount of work since older RPs have the potential to have more descendants than younger RPs. RPs may

be transferred several times before being instantiated; hence, arguments are transferred only after the RP has been instantiated by a host VG.

To further reduce the overhead associated with load distribution, the load balancing algorithm is designed to require a minimum of communication among the RP Managers in the FTPP. Whenever a change in its local load has occurred, the RP Manager broadcasts a message containing the number of RPs in its balance and ready queues. Other RP Managers use these messages to update their view of the load in the system. This locally maintained data is the basis for making RP transfer decisions. RPs are transferred only when the local load is greater than the system average and then only to a VG whose load is below that of the system average. This algorithm provides the system with some hysteresis, which stabilizes the system by reducing overshoot and oscillation. This simple algorithm results in a very short amount of processing time spent in executing load balancing logic. Furthermore, once the decision is made to transfer an RP, no further handshaking is required. The RP Manager sends an imperative "transfer load" message to its counterpart on a remote VG, who adds this RP to its balance queue if it has a free entry in its execution list. Otherwise, the recipient immediately sends the RP to another VG and begins checking for deadlock.

To support kill and graceful degradation algorithms, the VG accepting the RP sends an acknowledgement to the VG which is hosting the RP's parent. This maintains the link between the parent and the child RPs.

Load balancing is an integral part of graceful degradation and deadlock recovery. When the redundancy of a triplex or quadruplex VG is reduced due to a failure, it can still safely participate in error recovery strategies if those strategies can effect reconfiguration quickly enough to keep the probability of a second failure during reconfiguration sufficiently low. Graceful degradation exploits this fact. For deadlock recovery, all RPs in the balance queues can be immediately terminated, thus freeing up resources for the branch of the computation which is to be allowed to advance to its leaves (See [Tr87]).

2.5.4 LOCAL RP INSTANTIATION AND SCHEDULING

To instantiate an RP, the RP Manager obtains memory for its arguments and responses and creates a task to execute the RP. The RP chosen for local instantiation is taken from the tail of the balance queue. This is part of a deadlock avoidance strategy; since it is possible that parents are waiting for their children to complete, preferentially instantiating "younger" RPs for execution promotes earlier completion of the computation.

2.5.5 RP KILL PROTOCOL

Several situations arise in which RPs must be terminated. For example, a parent may no longer require the result of a child, as in the A* example. Children are also terminated to prune a branch during deadlock recovery. Finally, one means of graceful degradation is simply to kill all descendants of RPs executing on a degraded VG. A desirable characteristic of a kill protocol is speed of execution to prevent RPs that are to be killed from spawning further RPs which must in turn be killed recursively. In our implementation the kill protocol is initiated through the calls `rp_release()` and `rp_flush()`. An RP is killed by removing it from the execution and pending lists. The entries are restored to their respective free lists and any memory allocated for the use of the terminated RP is deallocated. If a task has been started for this RP, it is stopped.

The kill protocol originates with the RP Manager on the VG hosting the parent RP. If the target RP is on the local execution list, the actions described above are simply carried out, and the kill is complete. However, if the RP has been transferred to a remote VG, the status field of the target's pending list entry is marked KILL and a kill message is sent to the RP Manager of the VG which sent the most recent transfer acknowledgement to the parent. Should a transfer acknowledgement arrive for an RP which has a pending entry status of KILL, implying another transfer of the target RP, a kill message is forwarded to that VG. When the kill message is received, the children of the target are flushed, thereby propagating the kill, the target RP is removed from the execution list, and memory and task deallocations are made. If the target RP is no longer present on this VG, the kill message is ignored. Finally, a kill acknowledgement is returned to the RP Manager issuing the kill request. When a kill acknowledgement message arrives at the parent's VG, the pending entry and associated RP identifier are released by the RP Manager. If child responses are returned for an RP with a pending entry status of KILL, they are discarded.

3. GRACEFUL DEGRADATION ALGORITHMS

One of the primary purposes of the functional programming model is to facilitate Graceful Degradation (GD) strategies. GD is responsible for halting the RPs on a Degraded Virtual Group (DVG) and reestablishing them on reliable VGs. Generally desirable features of any GD algorithm are fast execution time and minimal use of system resources during execution. On a system consisting of reconfigurable redundant groups such as the FTPP, GD execution time has a direct impact on system reliability. Since it is the occurrence of a fault

in the VG that triggers GD, quick action is required before a second fault can occur in that DVG, defeat the voting in a triplex or quadruplex VG, and possibly cause system failure.

GD activity is triggered by a message from an FDI task running either locally or on another VG. The message indicates that a particular VG has suffered a permanent fault and must evacuate its load. This Degraded Virtual Group receives this message and, because it can continue to function temporarily in the presence of a single fault[3], initiates the GD algorithm. After completion of GD, the RP Manager signals FDI that the DVG is idle and available for repair or dispersal. Three different GD algorithms were developed and implemented in order to compare their performance under various workloads. These algorithms are discussed in the following three sections.

3.1 "RUN TO COMPLETION"

In the simplest GD algorithm, the DVG immediately sets its load to its maximum value when it receives a "degrade" message from FDI. This results in the rapid offloading of all migratable RPs, while blocking the arrival of additional RPs from other VGs. The DVG then executes locally instantiated RPs as usual, except that all locally spawned children are immediately balanced away. GD is complete when all locally instantiated RPs complete. This algorithm does not result in the quickest completion of GD, but is useful as a benchmark for other designs. The other two GD algorithms developed reduce the time deficiency in different ways.

3.2 KILL AND RESTART

The fastest GD algorithm developed to date exploits the Kill and Restart capabilities of the functional model. This algorithm is "fastest" in the sense that it completes GD in the shortest amount of time. Upon receipt of a "degrade" message, the DVG initiates a kill protocol on all locally instantiated RPs. In addition, for each of these RPs killed, a "restart" message is sent to its parent (unless the parent is on the DVG). Parents receiving restart messages attempt to re-spawn the child. After all locally instantiated RPs are killed, any RPs on the balance queue are transferred to reliable VGs.

Although Kill and Restart is the fastest of the three algorithms, as measured by the work performed by the RP Manager to effect the graceful degradation phase of the fault recovery process, it wastes systems resources. Killing several branches of a computation tree

[3]This discussion assumes that VGs are of triplex redundancy or greater.

in this manner can be extremely inefficient, especially if the DVG contains RPs near the root of the tree. If the DVG hosts the root of the computation tree, not only is the whole tree killed, but difficulty may arise in restarting it. Thus, while the system may roll back to a stable configuration quickly, the application program may suffer a severe performance penalty in terms of the time it takes to complete a computation. These difficulties inspired the development of our third GD algorithm, which saves all computation in progress on the DVG.

3.3 COMPUTATION SAVING GRACEFUL DEGRADATION

Computation Saving Graceful Degradation (CSGD) conserves system resources over other GD algorithms by moving each instantiated RP to a reliable VG, relinking the RP with its parent and children, and restarting it. We have chosen this algorithm for study over other designs which save previous computation (such as grandparent splicing[4] [Li86]) for several reasons. Because our system supports dynamic load balancing, it is difficult to determine appropriate length timeouts for RP responses to messages. In addition, since GD is expected to be infrequent, it is inefficient for RP Managers on nonfaulty VGs to have to account for DVGs. Requiring such accounting would reduce the modularity of the system implementation as well.

CSGD suspends scheduling of RPs on all VGs in the system upon reception of the "degrade" message. The message ordering properties of the FTPP ensure that all RP Managers perceive identical ordering of the "degrade" message with respect to normal RP message traffic, and therefore take mutually consistent actions. This "freeze" allows the RP Manager of the DVG to obtain consistent load information from each VG and to execute without having to consider normal RP message traffic. It also allows all RP Managers to devote their full attention to GD, thereby decreasing execution time.

CSGD then begins the RP transfer and relink portion of GD. Transfer begins with the oldest instantiated RP on the DVG. The reason for choosing the oldest instantiated VG is to ensure that an RP whose parent was on the DVG has its parent's new location after the parent has moved. The specific function name and generation number of the RP are sent to an appropriate VG. Upon receipt of this information, that VG's RP Manager creates an execu-

---

[4] Upon processor failure an applicative tree may be partitioned into several pieces. When a parent discovers the failure of a child task, the parent generates a twin of the faulty child which inherits all offspring of the faulty task. The necessary linkages from the children to the parent of the faulty task are maintained via their grandfather pointers, which point from each task to its ancestor in the grandfather processor.

tion list entry for the RP, sends the RP's new execution list entry identifier to the RP's pending list entry on its parent's virtual group, and waits for the pending list information of the moved RP's children. This pending list information is sent by the DVG and consists of the execution list entry identifier of the child and the response of the child (if it has completed). Children of a transferred RP receive notification of its new host VG when their new pending list entries are created there.

After all instantiated RPs are moved to new locations and restarted, the RPs on the DVG's balance queue are transferred to reliable VGs. The "freeze" on RP execution is removed and RP Manager execution is resumed. However, after an RP transferred during CSGD restarts, any `rp_create()` and `rp_start()` calls that it performs are ignored until it has attempted to recreate all the children it created on the DVG prior to CSGD. These children have been created already and are running normally elsewhere so it links up with them instead of creating new ones. After all these repeated creations have been attempted by the transferred RP, normal creation of children is enabled. We chose the restart method of restoring a transferred RP's state rather than one which freezes and transfers the RP's entire state as it existed at the time of CSGD, because our method decreases the size of the GD messages and reduces the algorithm's complexity. After all RPs originating on the DVG have restarted and are able to create new children, CSGD has completed.

Of the three designs which were implemented and evaluated, our initial experience indicates that Computation Saving Graceful Degradation has the most desirable characteristics. CSGD completes quickly while saving previous computation, keeps message size to a minimum, and preserves modularity. However, more conclusive results will require comparative evaluation in the context of a given application.

4. APPLICABILITY OF THE FUNCTIONAL PROGRAMMING MODEL TO THE ACTIVATION FRAMEWORK INTELLIGENT SYSTEM PARADIGM

The Activation Framework (AF) knowledge-based system paradigm has been proposed for real-time intelligent systems [Gr85]. Under the current contract a small effort was authorized to evaluate the mutual compatibility of the AF paradigm and the functional programming model described above.

An application in AF consists of a network of communicating experts, each of which is called an Activation Framework Object (AFO). Multiple AFO networks may reside in a processing system. Discussion of generation of the AFO networks is beyond the scope of this overview. AFOs possess state which persists over the lifetime of the computation.

AFOs communicate via message-passing; reception of a message by a destination AFO activates a computation by that AFO, during which it may transmit one or more messages to other AFOs, change its persistent state, suspend its execution, and perform other object-based functions. Messages are ASCII strings, and are addressed to a given destination AFO by a system-wide unique "AFO name." Parallelism is achieved in the AF paradigm by allowing many AFOs to execute simultaneously in a distributed or parallel environment. Mapping of AFOs to processors in a distributed system is implemented via a look-up function from AFO name to destination processor and task identifier. Currently, this mapping is static and determined at initiation of the computation.

The functional programming model makes radically different assumptions about the structure of the computation. As an example, we compare the fundamental unit of parallel activity in our functional programming model, the Remote Procedure (RP), to its counterpart in AF, the Activation Frame Object (AFO). The RP has no mutable persistent state, the computation it performs being fully described by its input arguments and the destination of the result. It may not send any messages to another RP other than the computational result message to its parent, or generation and parallelism control messages to its children. Its only side effect is the generation of a computational result which it returns to its parent; it cannot change any other global state. The AFO, on the other hand, is by definition a long-lived object possessing mutable persistent state. The computation it performs at a given time is a function of its entire input message history, its initial condition, and, consequently, its internal state. It may at any time transmit messages to any destination AFO of which it knows the AFO name, and it can change the computation's global state, of which its internal state is a part.

From our evaluations it appears that the RP model is appropriate for applications which are highly structured (although not necessarily of predetermined size and intensity), regular, and algorithmically intensive. The A* and other regular search algorithms appear to be archetypes of this class of computation. The AF model appears to be more appropriate for applications where much of the knowledge is heuristic, non-algorithmic, and occasionally inconsistent, the system must maintain an ongoing estimate of the global state of the computation, heterogeneous parallel modules must coexist, and rapid changes of computational focus must be supported. It would also probably be easier to capture the knowledge into an AF-type system than an RP-type system because the former does not constrain the programmer to algorithmically capture the whole of the computation; instead the programmer may specify a piece of it at a time without worrying whether it is even consistent with the

remainder of the application's knowledge. Consequently it is possible to conclude that, at a superficial level, the AF and RP models are incompatible. However, in our opinion there may be mutually synergistic concepts.

For example, the AF model presumes that AFOs are load balanced. Static load balancers have been developed based on estimated AFO computational requirements and inter-AFO message density. However, if these estimates turn out to be inaccurate in practice, or if faults necessitate evacuation of the AFOs resident on a degraded VG, then a dynamic load balancing technique similar to that developed under this task may be required.

As another example, the AF concept appears to implicitly assume that the execution of a given AFO is inherently serial[5]. This may not necessarily be the case, and in fact may lead to less-than-optimal speedup. For example, the Event Diagnosis module of the Advanced Tactical Navigator is expected to require about six AFOs. If there are roughly five other modules in the ATN application and their AFO suite is of commensurate cardinality, then the total number of ATN AFOs is roughly thirty six, which could at best keep thirty six processors busy[6]. Although it can be argued that in this case one could augment the functionality assigned to the parallel processor, it is reasonable to inquire whether greater speedup would be obtained if AFOs themselves can be parallelized, and whether a RP-based scheme would be appropriate for internal AFO execution.

The functional programming model can benefit from AF concepts as well. One of the drawbacks of the current functional programming model's implementation is that the programmer must manually create and control the parallel execution of the RPs. If a way could be found to automatically convert appropriately structured EFGs into applicative functional trees, this problem would be alleviated. It would appear that the code generator being developed under the Knowledge Representations into Ada Methodologies (KRAM) program could be augmented to optionally generate RP trees when the EFG is of an appropriate structure, which could execute internal to one or more AFOs or in parallel with an AFO network. This approach could expand the applicability of the "toolkit" approach being developed under KRAM to another class of parallel computation.

---

[5]If an Evidence Flow Graph (EFG) is "compiled" with a given number of processors in mind, the number of AFOs generated could be a function of the number of processors in the target machine. Alternatively, the EFG could be used to identify parallelism internal to an AFO.

[6]If the application generates a small number of AFOs, each one must be very computationally intensive with respect to time spent sending messages, otherwise quite low processor utilizations will result.

5. CONCLUSIONS

A functional Remote Procedure-based programming model has been designed and implemented on the Draper Fault Tolerant Parallel Processor. This model presents a simple and powerful interface to the application programmer, facilitating creation and control of parallel algorithms while masking the distributed, fault-tolerant, and highly reconfigurable nature of the FTPP architecture. Our initial experience in programming simple applications has borne this out. In addition, the model has facilitated the development of load balancing, graceful degradation, and deadlock recovery algorithms. The first two of these algorithms have been implemented and successfully demonstrated.

In this sense the functional programming model has lived up to its expectations. However, programmers we have talked to are unsure whether they would be able to adequately express their applications in a side effect-free applicative tree of functional RPs. We are therefore of the opinion that, regardless of its desirable fault tolerance-related features, for a programming model such as the one we have developed to come into widespread use, a software development methodology such as that being developed under the Knowledge Representations into Ada Methodologies (KRAM) program is needed. Such an approach would allow the application programmer to express the algorithm or knowledge in a form which is convenient to use (e.g., production rules), semantically congruent to the application itself, and validatable at some level of instantiation, followed by a more or less automatic conversion from the high-level representation to the implementation language. Even more attractive would be the capability to appropriately mix functional and object-oriented programming models within an integrated application development environment.

6. REFERENCES

[Ab88] Abler, T. A., "A Network Element-Based Fault Tolerant Processor," S. M. Thesis, Massachusetts Institute of Technology, June 1988.

[Fa85] Fasel, J. H., Douglass, R. J., Michelsen, R., and Hudak, P., "A Distributed Implementation of Functional Program Evaluation," Los Alamos National Laboratory, New Mexico, Department of Energy Contractor Report DE85009573, 1985.

[Gr85] Green, P. E., "The Activation Frame Method for Real-Time Expert Systems," Technical Report EE85PG04, Department of Electrical Engineering, Worcester Polytechnic Institute, Worcester, MA, October 12, 1985.

[Ha87] Harper, R. E., "Critical Issues in Ultra-Reliable Parallel Processing," PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.

[Ho83] Hosseini, S. H., Kuhl, J. G., and Reddy, S. M., "An Integrated Approach to Error Recovery in Distributed Computing Systems," *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, June 1983, pp. 56-63.

[Ja86] Jagannathan, R., Ashcroft, E. A., "Fault Tolerant Aspects of the Eduction Model and Architecture," *Proceedings of the IEEE/AIAA 7th Digital Avionics Systems Conference*, pp. 515-22, 1986 .

[Jo85] Jones, H. L., Pisano, A. L., "Adaptive Tactical Navigation," Report No. AFWAL-TR-851015, The Analytical Sciences Corporation, Reading, MA, April 1985.

[Ki85] Kilber, D .F., and Conery, J., "Parallelism in AI Programs," *Ninth International Joint Conference on Artificial Intelligence*, August 1985.

[Ku86] Kuhl, J. G., Reddy, S. M., "Fault Tolerance Considerations in Large, Multiple-Processor Systems," *Computer*, vol. 19, no. 3, pp. 56-67, March, 1986.

[Ku88] Kumar, V., Ramesh, K., and Rao, V. N., "Parallel Best-First Search of State-Space Graphs: A Summary of Results," *Proceedings of the National Conference on Artificial Intelligence*, AAAI-88.

[La86] Lala, J. H., "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications," *16th Annual International Symposium on Fault Tolerant Computing Systems*, Vienna, Austria, 1-4 July 1986.

[Li86] Lin, F. C. H., Keller, R. M., "Distributed Recovery in Applicative Systems," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 405-12, 1986.

[NA86] National Aeronautics and Space Administration Project Plan, "System Autonomy Demonstration of Thermal Management for Space Station", Ames Research Center, September 1986.

[Pe84] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Reading, Massachusetts: Addison-Wesley, 1984.

[Tr87] Troxel, G. D., "Detection and Recovery from Deadlock in a System using Remote Procedure Calls," S. B. Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.

[Ve84] Vegdahl, S.R., "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Transactions on Computers*, vol C-33, no. 12, pp. 1050-71, 1984.

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181938		

4. Title and Subtitle
Investigation of the Applicability of a Functional Programming Model to Fault-Tolerant Parallel Processing for Knowledge-Based Systems

5. Report Date

6. Performing Organization Code

7. Author(s)
Richard Harper

8. Performing Organization Report No.

10. Work Unit No.
549-03-31-03

9. Performing Organization Name and Address
The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, MA 02139

11. Contract or Grant No.
NAS1-18565

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

13. Type of Report and Period Covered
Contractor Report

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor: Sally C. Johnson

Final Report for Task 4

16. Abstract

In a fault-tolerant parallel computer, a functional programming model can facilitate distributed checkpointing, error recovery, load balancing, and graceful degradation. Such a model has been implemented on the Draper Fault-Tolerant Parallel Processor (FTPP). When used in conjunction with the FTPP's fault detection and masking capabilities, this implementation results in a graceful degradation of system performance after faults. Three graceful degradation algorithms have been implemented and are presented. A user interface has been implemented which requires minimal cognitive overhead by the application programmer, masking such complexities as the system's redundancy, distributed nature, variable complement of processing resources, load balancing, fault occurrence and recovery. This user interface is described and its use demonstrated. The applicability of the functional programming style to the Activation Framework, a paradigm for intelligent systems, is then briefly described.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel Processing Fault Tolerance Computer Architectures Expert Systems Knowledge-Based Systems	Unclassified-Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified		